Initiation à l'algorithmique et au langage Ruby

Jean-Marc DESBONNEZ

Collège Sainte-Marie — Mouscron

Septembre 2014

1 Généralités

1.1 Quoi ?

Le mot *algorithme* est d'origine arabe ; le mathématicien *Muhammad Ibn Musa al Kharzami* (début du IXe siècle) utilisait des ensembles de symboles et de procédés de calculs mathématiques.

A l'origine, un algorithme est donc un ensemble de règles opératoires propres à un calcul. Certains algorithmes sont célèbres : calcul d'une racine carrée (Héron d'Alexandrie), calcul de décimales du nombre π, algorithme d'Euclide (300 av. J.C.) pour le calcul du P.G.C.D. (1) de 2 nombres, etc.
Ces règles opératoires consistent à décomposer un calcul complexe en un enchaînement de calculs élémentaires.

De tels procédés de calculs sont utilisés dans les calculettes et les ordinateurs, c'est à dire dans des machines capables uniquement de faire des calculs élémentaires en langage binaire. A l'ère de l'informatique, on fait faire par un ordinateur presque n'importe quoi ; il « suffit » d'écrire l'enchaînement des actions nécessaires à l'accomplissement d'une tâche donnée, l'ordinateur jouant le rôle de *robot bête et discipliné* (mais rapide ...).

L'algorithmique consiste à décomposer un problème complexe en une succession de tâches élémentaires.

C'est « l'art du saucissonage ».

1.2 Exemple

Dans la vie de tous les jours, nous côtoyons beaucoup d'algorithmes : n'importe quel « mode d'emploi », de la machine à lessiver au lecteur DVD en passant par le GSM, est finalement un algorithme, puisqu'il consiste en une succession d'instructions simples que l'utilisateur final est sensé devoir (savoir) accomplir.

Détaillons par exemple une banale promenade à vélo :

un humain ne va (heureusement) pas faire ce genre de raisonnement, et tous ces gestes seront opérés de manière naturelle, mais s'il faut mettre un robot sur le vélo, il faudra bien *lui dire* ce qu'il doit faire !

(1). Plus Grand Commun Diviseur

Début
Mettre une tenue adéquate
Prendre le vélo
Si problèmes techniques
    Alors réparer
Monter sur le vélo
Tant que non arrivé à destination
    Pédaler
    Si montée ou descente
        Alors changer de vitesse
    Se reposer un peu
.../...

.../...
Remonter sur le vélo
Tant que non arrivé à destination
    Pédaler
    Si montée ou descente
        Alors changer de vitesse
Prendre une douche
Fin

Il est même toujours possible de détailler certaines actions de manière encore plus fine :

Si problèmes techniques
    Selon que
        Pneu dégonflé
            Regonfler
        Selle trop haute
            Baisser selle
        Selle trop basse
            Lever selle
        Chaîne rouillée
            Graisser chaîne
        Etc.

Et pour pousser le bouchon un peu plus loin encore :

Selle trop haute :
    Prendre clé
    Desserrer écrou
    Tant que pas à la hauteur
        Baisser
    Serrer écrou
    Ranger clé

A ce rythme-là, on constate qu'il n'y a plus de fin !
L'important est de savoir à quel niveau de détails il faut s'arrêter. C'est ce que l'on appelle les actions de base, ou, dans le jargon informatique, les « primitives du langage ».

1.3 Informatiquement parlant

En informatique, l'algorithmique est l'étape préliminaire (et indispensable) à la réalisation de *programmes*, aussi appelés *logiciels* ou encore *macro-instruction* dans certains langages de programmation.

Un programme est une succession d'instructions élémentaires écrites dans un langage qui sera interprété et exécuté par la machine.
Avant la phase de traduction dans un langage de programmation, il est nécessaire de réaliser un « plan » dans lequel on tiendra compte de toutes les éventualités qui pourraient se présenter, des erreurs possibles dues à l'utilisateur, etc.
Faire ce plan, c'est écrire l'algorithme du problème. C'est la phase la plus complexe, car il faut penser à tout ; la phase de traduction n'est que technique, il suffit d'appliquer quelques *règles de grammaire*, la *syntaxe* du langage de programmation.

L'algorithme doit être, en principe, **indépendant** de tout langage de programmation.

On peut comparer la réalisation d'un programme à la construction d'une maison : il faut d'abord que l'architecte fasse les plans, et ensuite la phase de construction par l'entrepreneur peut commencer.
En principe, les plans doivent être compréhensibles par n'importe quel entrepreneur, de n'importe quelle région, pour autant évidemment que le terrain soit conforme.
Dans l'écriture des algorithmes, on utilise fréquemment des TESTS, encore appelés CONDITIONS ; ce sont des expressions qui sont soit vraies, soit fausses, et qui orientent la suite des opérations.
C'est l'*algèbre de Boole* qui régit les lois concernant ces expressions, aussi appelées EXPRESSIONS BOOLÉENNES, du nom du mathématicien (2) qui s'y appliqua.

1.4 Schéma de travail

Ce schéma de travail est INDISPENSABLE à la création d'un « logiciel », quel qu'il soit.
Au départ, un problème est posé, par exemple « calculer la somme des 50 premiers nombres pairs ».
À partir du problème, il faut en écrire l'algorithme, c'est-à-dire la succession des actions élémentaires qui vont aboutir à la solution du problème. Cette succession d'actions élémentaires est écrite dans un *langage structuré*, appelé *pseudo-code*, proche des langages informatiques modernes, tout en restant le plus indépendant possible d'un langage particulier.
L'étape suivante, technique, est la traduction de l'algorithme dans un langage informatique approprié au problème de départ.
Le résultat de la traduction de l'algorithme dans un langage de programmation est appelé code source ; il est réalisé par l'intermédiaire d'un éditeur de textes spécialisé.
Ce code source sera ensuite *transformé* en code exécutable via un compilateur associé au langage de programmation. C'est ce code exécutable qui est appelé *programme*, *logiciel*, ou *macro instruction*, ...

On verra par la suite que le langage *pseudo-code* est un langage très simple, avec peu d'instructions et de structures, et donc assez simple à mettre en oeuvre.
Il est calqué sur la plupart des langages de programmation modernes, ce qui le rend très « universel ».

Au début de l'informatique, un algorithme consistait en un plan (3) composé de figures géométriques ayant chacune une signification bien précise, et reliées par des flèches indiquant la suite des calculs à effectuer. Cette manière de voir ne convient plus aux langages actuels.

(2). George Boole, 1815-1864.
(3). Appelé *ordinogramme*.

1.5 Un exemple concret

Pour donner au lecteur l'envie de lire la suite, voici **un** (on pourrait en imaginer d'autres) algorithme qui permet de calculer la somme des 50 premiers nombres pairs.

1.5.1 Pseudo-code et traduction en Ruby

Pseudo-code

```
somme ← 0
compteur ← 1
tant que compteur < 50
        somme ← somme +2*compteur
        compteur ← compteur + 1
fin du tant
écrire somme
```

Traduction en langage *Ruby* (4) :

```
somme=0
compteur=1
while compteur<50
        somme=somme+2*compteur
        compteur=compteur+1
end
puts "la somme vaut : #{somme}"
```

Et la réponse est : 2450 ☺

1.5.2 Un peu d'explications, quand-même

A ce stade, elles tombent du ciel ; de plus amples explications suivront ...

somme, compteur : ce sont des *variables*, espaces-mémoire identifiés par un nom.

← opération d'*affectation*, qui consiste à mettre un contenu dans une variable ; dans beaucoup de langages de programmation, elle est traduite par le signe « = ».

somme ← somme+2*compteur : on prend le contenu de la variable *somme*, on y ajoute le double de la variable *compteur* ; le résultat est ré-affecté à la variable *somme*.

compteur ← compteur+1 : idem que ci-dessus : on prend le contenu de la variable *compteur*, on y ajoute 1 ; le résultat est ré-affecté à la variable *compteur*.
Cette opération est appelée *incrémentation* de la variable *compteur*.

tant que : c'est une des structures de type *répétitive* ; toutes les instructions de la structure seront exécutées tant que la condition d'arrêt n'est pas vérifiée ; logiquement traduite par « *while* ».

écrire somme : consiste à afficher (à l'écran) le contenu de la variable *somme* ; traduction en Ruby par l'instruction « puts ».

(4). Langage conçu en 1993 par l'informaticien japonais Yukihiro Matsumoto, surnommé *Matz*.

2 Ruby, un langage de programmation

Faire de l'algorithmique sans utiliser un langage de programmation c'est un peu faire du solfège sans jouer d'un instrument : c'est frustrant !
Rien de tel en effet pour savoir si l'algorithme fonctionne bien que de « faire jouer la partition » par l'ordinateur lui-même.

On sera donc amené à courir deux lièvres simultanément :

- ➙ L'algorithmique.
- ➙ Le langage de programmation.

2.1 Langage de programmation

2.1.1 Rôle

Traduction logique : un langage de programmation est un langage dans lequel on écrit des programmes (ensembles d'instructions exécutables par un ordinateur).

Au départ (et à l'arrivée aussi d'ailleurs), le seul langage exécutable par un ordinateur est le *langage binaire* : langue extrêmement basique, puisque composée de 2 « mots » uniquement, 0 et 1.
Pour un humain, il est assez indigeste ; à essayer de retaper sans se tromper :

1010010011101011001010100110010101110110101101110101001001010 10

ou d'écrire aisément le mot « MATH » sachant que le code binaire du A est 01000001 [(5)], celui du H est 01001000, celui du M est 01001101 et celui du T est 01010100.

Pour surmonter cette difficulté, les informaticiens ont mis au point ce qu'on appelle des *langages de haut niveau.*

Ce sont des langages techniques, dont le vocabulaire est relativement « humain », avec des mots à consonance anglaise (universalité oblige).

De manière imagée, on peut dire qu'ils fonctionnent comme un dictionnaire *humain-binaire*, à savoir qu'ils traduisent en binaire des instructions écrites en « humain » afin de permettre leur exécution par une machine qui ne comprend que le binaire.

Le codage en binaire est appelé *compilation.*

2.1.2 Lequel choisir ?

Il existe de (très) nombreux langages de programmation ; ils ont bien sûr beaucoup évolué, en même temps que l'informatique ; ils ont parfois un domaine d'application particulier : plutôt internet, plutôt scientifique, plutôt gestion, . . . , et sont aussi sujets au phénomène de mode.

(5). Code ASCII, American Standart Code for Information Interchange.

Certains sont aussi connus que les dialectes du fond de l'Amazonie, d'autres font plus parler d'eux . . . .

Pour ne citer (chronologiquement) que ceux auxquels je me suis frotté : FORTRAN, ASSEMBLEUR, BASIC, TURBO BASIC, PASCAL, CLIPPER, VISUAL BASIC, APPLESCRIPT, C, JAVA.

Certains sont plus faciles à apprendre que d'autres (l'*anglais* est plus facile à apprendre que le *chinois*), mais dans tous les cas, c'est 1 heure de théorie pour 24 heures de pratique . . . c'est un métier.

2.1.3 Ruby !

Sans entrer dans tous les détails, pourquoi ce choix ?

- Pour un apprentissage de base de la programmation, il est facile.
- Pour une utilisation approfondie, il est puissant et performant.
- Il est gratuit.
- Il est facile à installer sur l'ordinateur (d'ailleurs sur un MAC il est déjà préinstallé) !
- Il permet une utilisation *interactive* : on tape une instruction, elle est immédiatement exécutée, idéal pour la découverte.

2.1.4 Installation et mise en place sous Windows

Ruby peut être téléchargé à l'adresse

`http://rubyinstaller.org`

Si nécessaire, le lecteur trouvera également des informations sur le site [6]

`http://www.div-math.fr`

à la rubrique « Annexes sur l'installation de Ruby ».

2.2 Ruby interactif

L'une des raisons principales du choix de *Ruby* comme langage de programmation est que l'on peut l'utiliser en *mode interactif*, à savoir que dès qu'une instruction est encodée, elle est exécutée.

Bien évidemment, ce mode a des limites, et après avoir pris en main les instructions de base, il suffira de passer en *mode éditeur*.

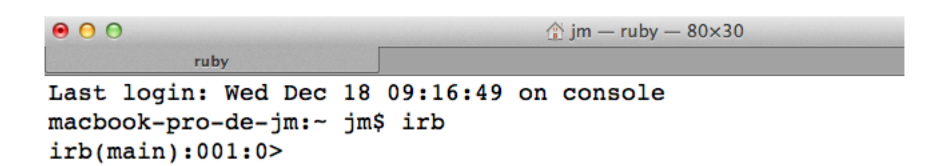
(6). *Divertissements mathématiques et informatiques*, Laurent Signac, édition Minimax.

2.2.1 Activation sur Mac (sous OsX)

Ruby étant déjà pré-installé, il suffit d'activer son environnement :

(1) Exécuter l'application TERMINAL.APP
dans le dossier Applications > Utilitaires
Pour une utilisation courante, il est judicieux de placer l'application dans le dock où 1 clic suffit pour l'exécution.

(2) Au message d'invite, taper l'instruction *irb* [7] puis Enter .



Et hop, on peut commencer !
En fait, on peut commencer en mode *interactif* ; on verra plus loin que pour travailler en mode *programmation*, il faudra d'abord activer, via le terminal, le répertoire de travail qui contient les fichiers de commandes (fichiers source).

2.2.2 Activation sur Pc (sous Windows ou Linux)

Voir point 2.1.4 page 6.

2.2.3 Les opérations arithmétiques de base

Il y en a 6 : addition(+), soustraction (-), multiplication (*), division (/), exponentiation (**) et modulo (%).
A essayer successivement ...
$8 + 9.41 \rightsquigarrow$ (le séparateur décimal est le point)
$8 - 9$
$8 * 9$
$14/11 \rightsquigarrow 1$ (toute opération sur 2 entiers fournit un résultat entier, arrondi)
$14.0/11 \rightsquigarrow 1.2727272727272727$ (l'un des opérandes doit être réel pour un résultat réel)
$8 * *9 \rightsquigarrow 134217728$
$8 * *9 * *4 \rightsquigarrow$ désolé, pas assez de place ...
$432\%7 \rightsquigarrow 5 \quad \left(\frac{432}{7} = 61 \text{ reste } 5\right)$

(7). Interactif RuBy

2.2.4 Les calculs mathématiques un peu plus poussés

Fonction	Explication
`Math.sqrt(x)`	$\sqrt{x}$
`Math.cbrt(x)`	$\sqrt[3]{x}$
`Math.sin(x)`	sinus de x en radian
`Math.cos(x)`	cosinus de x en radian
`Math.tan(x)`	tangente de x en radian
`Math.exp(x)`	e^x
`Math.log(x)`	logarithme naturel (ou népérien) de x
`Math.log10(x)`	logarithme vulgaire (ou décimal) de x
`Math.log(x,a)`	logarithme en base a de x ($\log_a x$)
`Math::E`	le nombre d'Euler e=2.718281828459045...
`Math::PI`	π = 3.141592653589793 ...
`rand`	nombre réel aléatoire (18 déc) dans [0,1[
`rand(a)`	nombre entier aléatoire dans [0,a[
`rand(a..b)`	nombre entier aléatoire dans [a,b]

Table 1 – Opérateurs mathématiques de base

Attention : il est **impératif** de respecter la *casse* !

2.2.5 Mode interactif et algorithme

En mode interactif, on peut taper *successivement* les instructions d'un algorithme : taper par exemple celles permettant de calculer la somme des 50 premiers nombres pairs (voir instructions page 4).

L'encodage des instructions « ne produit rien », sauf la dernière, qui affiche le résultat.

Inconvénients majeurs de cette manière de procéder :

- Si on a fait une faute de frappe en cours de route, on le verra (peut-être) au résultat final, et il faut tout recommencer !
- Si on souhaite calculer la somme des 60 premiers nombres pairs, il faudra tout retaper, alors qu'il ne suffit que d'une petite modification à la ligne 3.
- Et pour des algorithmes de 4000 lignes ?

On voit ici les limites du mode interactif.

2.3 Ruby en mode Editeur

2.3.1 Editeur de textes

Le mode *éditeur* consiste à taper toutes les instructions de l'algorithme dans un document de type *texte*, à l'aide d'un logiciel appelé « éditeur de textes ». Les instructions ne sont pas interprétées (et donc pas exécutées) par Ruby au fur et à mesure de leur frappe ; c'est comme si on écrirait une « liste de courses ».

Ce document, appelé ***code source*** doit être enregistré sur le disque dur (attention, il y a des conventions pour le nom [(8)]) et pourra être exécuté par la suite au moyen d'une instruction via le Terminal.

Choix d'un éditeur

Il ne faut pas confondre « éditeur de textes » avec « traitement de textes ». Un éditeur de texte ne possède pas d'outils sophistiqués de mises en forme, de gestion de tableaux, de création de tables des matières, etc.

Pour taper des instructions, le minimum nécessaire est la numérotation et l'indentation des lignes (souvent activées par défaut) ; un plus bien appréciable est la coloration syntaxique des mots-clé du langage utilisé.

On trouve de nombreux éditeurs de textes gratuits sur internet ; chaque système d'exploitation en propose d'ailleurs un par défaut : *Textedit* sur MacOsX, *Wordpad* sur Windows.

Ils sont suffisants pour Ruby, mais ne permettent pas de numéroter les lignes (important pour de longs algorithmes) ni la coloration syntaxique du langage.
On utilisera ici *TextWrangler*, installé par défaut sur les nouveaux ordinateurs *cyberclasses*.

Par acquis de conscience, on peut toutefois vérifier certaines options :

TextWrangler > Preferences
- Editor Defaults : Auto-indent, Syntax coloring
- Languages : Ruby (Make Defaults) ; voir aussi *Options*
- Text printing : Print Lines numbers

Edit > Text options... > Display : Lines numbers

2.3.2 Enregistrement et exécution d'un programme

ENREGISTREMENT

Il est recommandé (et plus facile pour s'y retrouver) d'enregistrer le texte dans un **dossier** qui lui est propre.
Pour la démo, créer le dossier **ruby01** sur le **bureau** [(9)]
A l'aide de l'éditeur, taper les instructions (le code source de la page 4) et l'enregistrer sous le nom **som50pairs.rb** dans le dossier **ruby01**.

(8). Doit commencer par une lettre, ne contenir que des lettres et des chiffres, pas d'espace, pas de lettres accentuées, pas de signes de ponctuation, l'extension doit être **.rb**
(9). **bureau** est d'ailleurs lui-même un dossier (voir arborescence du disque) !

Exécution

Pour exécuter le programme source (c'est-à-dire exécuter toutes les instructions qu'il contient), il faut

1. lancer l'application *Terminal.app*.
2. activer le dossier **ruby01** du **bureau** à l'aide des *commandes Unix* appropriées (voir ci-après).
3. exécuter le programme source par l'instruction **ruby som50pairs.rb**.

Quelques commandes Unix indispensables

Unix est un système d'exploitation de premier niveau (sur lequel vient se superposer *MacOsX* [10]) et qui permet entre autre la gestion du disque dur via le *Terminal*.

> Pour la petite histoire, les bases de ce qui s'appellera *Unix* par la suite ont été développées pour de gros systèmes informatiques en 1969 par Ken Thompson (laboratoires de Bell), donc bien avant *MsDos* qui a été implanté sur les premiers « ordinateurs personnels » au début des années 1980.
> *Unix* étant plus sécurisé que *MsDos*, c'est une des raisons pour lesquelles les systèmes d'exploitation basés sur celui-ci sont moins sujets à des attaques virales que les systèmes d'exploitation basés sur l'autre.
> Fin de la petite histoire.

Pour activer un **dossier** (ou **répertoire**), **directory** en anglais, il est indispensable d'avoir une vue d'ensemble sur l'**organisation arborescente** du disque dur.

pwd	affiche le nom du répertoire actif
ls	affiche le contenu du répertoire actif
cd /	active le répertoire principal
cd rep	active le répertoire rep **fils** du répertoire actif (descente dans l'arborescence)
cd ..	active le répertoire **père** du répertoire actif (montée dans l'arborescence)

Si on a une vue d'ensemble de l'arborescence, on peut utiliser directement une instruction du type

cd /rep1/rep2/rep3

ce qui suppose que rep3 est un sous-répertoire de rep2 qui est lui-même un sous-répertoire de rep1 qui est un sous-répertoire du répertoire principal.

(10). Tout comme *Windows* se superpose à *MsDos*, MicroSoft Disk Operating System

3 Algorithmique et Ruby

Tout comme un maçon n'entreprend pas la construction d'une maison sans un plan de l'architecte, on n'entreprend pas la création d'un programme sans l'algorithme correspondant.

Dans ce chapitre, on mettra en place un *langage structuré* facilement transposable en un langage de programmation.

3.1 Variable et constante

Pour qu'une information soit traitée par un ordinateur, il faut qu'elle y soit mémorisée.

Une *variable* est un espace mémoire étiqueté (le nom de la variable) dont le contenu peut varier au cours des instructions.
Une *constante* est également un espace mémoire étiqueté, mais dont le contenu restera fixe (interdiction et impossibilité de modifier le contenu).

Les deux peuvent contenir entre autre un nombre, un texte (appelé CHAÎNE), etc. Dans le cas d'une variable, un ancien contenu sera remplacé par un nouveau (le suivant écrase le précédent).

Il y a deux types de variables :

— la variable **simple**, que l'on peut comparer à un tiroir.

— la variable **tableau**, que l'on peut comparer à une commode de plusieurs tiroirs.

3.2 Nom d'une variable et d'une constante (règles Ruby)

Ces règles peuvent toutefois changer d'un langage à l'autre.

Un nom de variable doit commencer par une lettre minuscule (ou le symbole *underscore*), peut contenir des lettres et des chiffres, pas de caractère accentué, ni d'espace ni de signes de ponctuation.

Un nom de constante doit commencer par une lettre majuscule, puis doit respecter les mêmes règles que les constantes.

Les tableaux seront traités ultérieurement.

3.3 Le langage structuré, en résumé

C'est un langage technique, simple, avec peu de syntaxe.

Il est composé de

— 3 instructions de base : l'affectation, la lecture et l'écriture (certains langages comme *Java* par exemple en nécessitent une quatrième, la déclaration)

— 3 structures : la séquence, l'alternative, la répétitive

3.4 La séquence

Une séquence d'instructions n'est rien d'autre qu'une suite d'instructions ; elles sont exécutées dans l'ordre dans lequel elles sont écrites (d'où la numérotation des lignes dans l'éditeur).

3.5 L'affectation

L'affectation consiste à mettre un contenu dans une variable ; le contenu peut être une donnée fixe, le résultat d'un calcul, le contenu d'une autre variable, etc.

langage structuré	Ruby
variable ← contenu	variable = contenu
prenom ← "jules"	prenom = "jules"
prix ← 20	prix = 20
taux ← 0.3	taux = 0.3
reduction ← taux * prix	reduction = taux * prix
prix ← prix − reduction	prix = prix − reduction

Avec l'*affectation*, c'est la personne qui écrit le programme qui « décide » du contenu de la variable (ne pas confondre avec la *lecture* où c'est l'utilisateur final qui décidera).

3.6 La lecture

La lecture est une instruction dite d'*entrée* ; elle permet de faire « entrer » une donnée de l'extérieur (l'utilisateur final, via le clavier) vers l'intérieur (une variable).

Il s'agit d'une « affectation de l'extérieur ».

La syntaxe Ruby dépend du type de contenu attendu : un nombre entier (integer), un nombre décimal (float), une chaîne de caractère.

langage structuré	Ruby
lire variable	variable = gets.to_i
	variable = gets.to_f
	variable = gets.chomp

Le suffixe **.chomp** a pour effet d'enlever le code de la touche Enter qui se place à la fin de la chaîne de caractères.
Essayer en mode direct les 2 instructions *a=gets* puis *a=gets.chomp*

Remarque importante

L'instruction de lecture **attend** que l'utilisateur encode une donnée via son clavier.

Encore faut-il qu'il sache ce qu'il doit encoder ! !

Il est donc indispensable que toute instruction de lecture soit précédée d'une instruction d'écriture : il faut au minimum poser une question ! !

3.7 L'écriture

L'écriture est une instruction dite de *sortie* ; elle permet de faire « sortir » un résultat vers l'extérieur, l'écran en ce qui nous concerne.

langage structuré	Ruby
écrire donnée	puts ...

La syntaxe de l'instruction *puts* dépend du type de donnée à écrire.

instruction	effet
puts prix	affiche le contenu de la variable *prix*.
puts "blabla"	affiche le texte "blabla".
puts "blabla #{prix}"	affiche le texte "blabla" suivi du contenu de la variable *prix* (si *prix* est de type numérique).
puts "blabla " + prix	affiche le texte "blabla" suivi du contenu de la variable *prix* (si *prix* est de type texte).
puts "blabla" + prix.to_s	affiche le texte "blabla" suivi du contenu de la variable *prix* (numérique) converti en chaîne de caractère (string).

3.8 Exercices

1. On encode 2 nombres ; calculer (et afficher, évidemment) leur somme, leur produit et la différence entre le premier et le second.
2. On encode les 3 coefficients d'une équation du second degré (celui de x^2 sera supposé non nul). Calculer le discriminant de l'équation.
3. On encode le rayon d'un cercle. Calculer sa circonférence et sa surface.
4. On encode la longueur des 2 côtés de l'angle droit d'un triangle rectangle. Calculer la longueur de son hypoténuse.
5. Permuter le contenu de 2 variables.

3.9 La structure alternative

C'est une structure qui permet de « prendre une décision » en fonction d'une condition soit vraie soit fausse.

Première forme, alternative simple

```
si condition
   instruction(s) cas vrai
sinon
   instruction(s) cas faux
fin
```

```
if condition
   instruction(s) cas vrai
else
   instruction(s) cas faux
end
```

Exemple

```
lire nombre
si nombre est pair
   écrire "nombre pair"
[ sinon
   écrire "nombre impair" ]
fin
```

```
puts "encoder un nombre entier positif"
nombre = gets.to_i
if nombre %2 == 0
   puts "#{nombre} est pair"
else
   puts "#{nombre} est impair"
end
```

Remarques

1. `[ sinon`
   `instruction(s) cas faux ]`
   est facultatif, et peut parfois ne pas exister.
2. Ne pas confondre le signe = opérateur d'affectation avec le signe == opérateur de comparaison dans une expression booléenne.

Le tableau qui suit reprend les différents opérateurs de comparaison.

Seconde forme, alternatives imbriquées

```
si cond1
   instruction(s)1
sinon si cond2
   instruction(s)2
sinon si cond3
   instruction(s)3
...
sinon
   autre instruction
fin
```

```
if cond1
   instruction(s)1
elsif cond2
   instruction(s)2
elsif cond3
   instruction(s)3
...
else
   autre instruction
end
```

Expressions booléennes

Une *expression booléenne*, encore appelée *condition*, est une expression dont la valeur est soit vrai (true) soit faux (false).

On les retrouve dans les structures alternatives, ainsi que dans les structures répétitives.

Le tableau qui suit reprend les valeurs de la disjonction et de la conjonction de 2 expressions booléennes, ainsi que de la négation d'une expression booléenne.

Expression	Explication
a == b	a est identique à b
a < b	a est strictement inférieur à b
a <= b	a est inférieur à b
a > b	a est strictement supérieur à b
a >= b	a est supérieur à b
a ! = b	a est différent de b
cond1 or cond2	cond1 ou cond2 (est fausse si les 2 sont fausses)
cond1 and cond2	cond1 et cond2 (est vraie si les 2 sont vraies)
not cond1	négation de cond1 (est vraie si cond1 est fausse, et vice-versa)

TABLE 2 – Opérateurs de comparaison et expressions booléennes

Troisième forme, alternative multiple

Elle est plus compacte que la première forme, et est préférable lorsqu'il y a plus de 2 cas à traiter.

```
selon que
   lorsque cond1 alors instr1
   lorsque cond2 alors instr2
   lorsque ...
   sinon autre instruction
fin
```

```
puts "encoder un nombre"
n=gets.to_f
case
   when (n<0) then puts "négatif"
   when (n==0) then puts "nul"
   else puts "strictement positif"
end
```

3.10 Exercices

1. On encode un nombre au clavier.
   — Calculer sa valeur absolue
   (il faudra se rappeler de la définition de valeur absolue ...).
   — Calculer son inverse (attention, zéro n'a pas d'inverse ...).
   — Calculer sa racine carrée (attention, un négatif n'a pas de racine carrée ...).
2. Encoder les 3 coefficients d'une équation du second degré, calculer le discriminant et les solutions éventuelles.
3. Ecrire un algorithme qui valide une heure (exprimée en heures, minutes et secondes) introduite au clavier sous la forme de 3 entiers.
4. Simuler le lancer d'un dé, et afficher le résultat sous la forme "un", "deux",..."six".
5. On encode le résultat R d'une interrogation (supposé compris entre 0 et 20). Calculer (et afficher) une appréciation selon la grille suivante :

Résultat	Appréciation
$\in [0, 5[$	nettement insuffisant
$\in [5, 9[$	insuffisant
$\in [9, 10[$	faible
$\in [10, 13[$	satisfaisant
$\in [13, 15[$	bien
$\in [15, 20]$	très bien

3.11 La structure répétitive, dite « boucle »

Une répétitive est une structure, qui comme son nom l'indique, permet d'exécuter plusieurs fois une ou plusieurs instructions, l'arrêt étant déterminé par une *expression booléenne*.

La règle fondamentale :

une répétitive DOIT s'arrêter !

Il est donc indispensable que parmi les instructions qui seront répétées, une au moins permette à l'expression booléenne de devenir soit *vraie* soit *fausse*, selon le type de répétitive.

Tous les langages de programmation proposent 2 types de répétitives :
— la répétitive *automatique* : le programmeur connaît à l'avance le nombre d'itérations.
— la répétitive *non automatique* : le nombre d'itérations dépend d'une condition.

3.11.1 La répétitive automatique

```
Pour compteur de debut à fin
   instruction(s)
fin
```

```
a=3
b=12
for k in a..b
   puts k.to_s
end
```

La variable *compteur* est initialisée à la valeur *debut* ; après chaque itération elle est incrémentée de 1 ; la répétitive s'arrête dès que la variable *compteur* a dépassé la valeur *fin*.

debut et *fin* doivent être des entiers (pas des flottants), et peuvent être négatifs.

Et bien évidemment, pour que la répétitive s'arrête, $debut \leqslant fin$!!

3.11.2 La répétitive non automatique

Ruby propose plusieurs formes :

```
Tant que condition
   instruction(s)
fin
```

```
k=3
while k<=12
   puts k.to_s
   k=k+1
end
```

```
Jusqu'à ce que condition
   instruction(s)
fin
```

```
k=3
until k>12
   puts k.to_s
   k=k+1
end
```

Les 2 exemples ci-dessus sont équivalents ; remarquer que les 2 conditions sont contraires.

Dans ces 2 types de répétitives, la condition est examinée avant l'exécution des instructions ; il se peut donc que les instructions ne soient jamais exécutées.

```
Répéter
   instruction(s)
tant que condition
```

```
k=3
puts k.to_s
begin
   k=k+1
   puts k.to_s
end while k<12
```

Dans ce type de répétitive, les instructions sont exécutées avant d'évaluer la condition.

Ce 3e exemple a été rendu équivalent aux 2 premiers.

3.12 Exercices

1. Validation de donnée
   - Forcer l'utilisateur à encoder au clavier un nombre non nul (un message d'erreur doit être affiché si le nombre encodé n'est pas valide).
   - Forcer l'utilisateur à encoder au clavier une réponse O ou N (O pour oui, N pour Non).
   - Forcer l'utilisateur à encoder au clavier un nombre entier positif inférieur à 100, multiple de 7 mais pas de 3.
2. Ecrire un algorithme qui permet à l'utilisateur d'introduire des nombres au clavier (arrêt lorsqu'il introduit la valeur 0) ; il faut alors afficher le nombre de valeurs introduites, à l'exception de la dernière nulle.
3. Même exercice que le précédent, mais afficher en plus la somme de toutes les valeurs.
4. Même exercice que le précédent, mais afficher en plus la valeur la plus petite et la valeur la plus grande.
5. Ecrire un algorithme qui permet de répondre à la question suivante :
   « si on additionne les nombres entiers consécutifs $1+2+3+\cdots$, à partir de quel entier la somme dépassera-t-elle 10000 ? »
6. Une partie de dé consiste à lancer 2 dés et à faire la somme des points obtenus. On joue 10000 parties et il faut calculer la répétition des sommes obtenues.
   Cet exercices aura plus d'intérêt avec les variables tableaux.